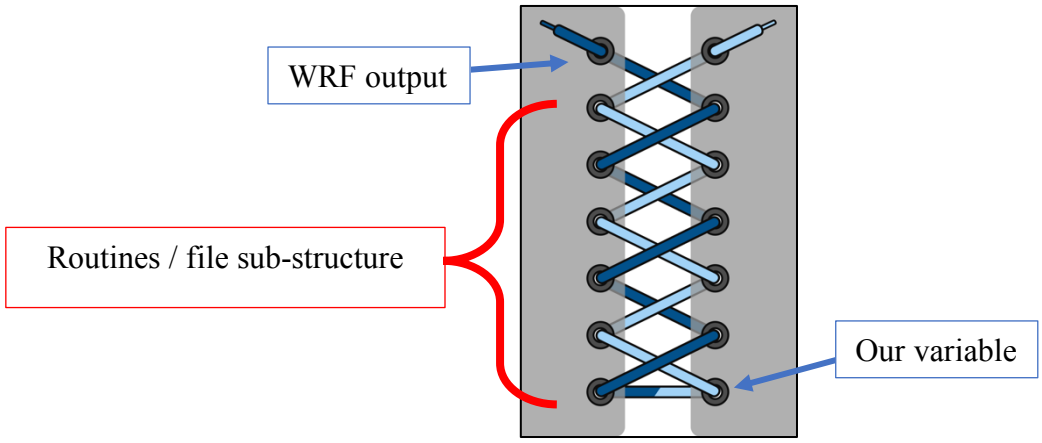
Extracting a Microphysical Variable from WRF

Extracting a variable in WRF is like lacing up a boot; the order of each layer is important and skipping a step will cause the entire thing to be incorrect. Indeed, it may feel as though you are threading a variable through the eye of a needle at times!



The thread is our variable that starts at the deepest level of the code hierarchy. The eyes are the steps of the code hierarchy that the argument must be fed through, at a broader level these are modules and within the modules they are subroutines.

1. Subroutines

In the WRF source code the process of threading the variable is complicated by the use of subroutines, which are similar to functions in python. Understanding Fortran subroutines will make this whole process a lot easier, so please do not skip this part!

Here is an example function in python and Fortran. It does the same thing in each case:

```python
def Function_Example(input_one, input_two, opts=None):
    """Print an opening statement, multiply arguments.
    If third argument present, add to total
    """

    print("A function has begun!")

    output = input_one*input_two
    if opts:
        output+=opts

    return output

Function_Example(2, 3)
```

```fortran
SUBROUTINE Function_Example(input_one, input_two, opts, output)
    REAL :: input_one, input_two, output
    REAL, OPTIONAL :: opts

    PRINT *,'A function has begun!'
    output = input_one*input_two

    IF( PRESENT(opts)) THEN
        output = output + opts
    ENDIF

    RETURN
    END SUBROUTINE Function_Example


#call subroutine
REAL :: output
CALL Function_Example(2, 3, output)
```

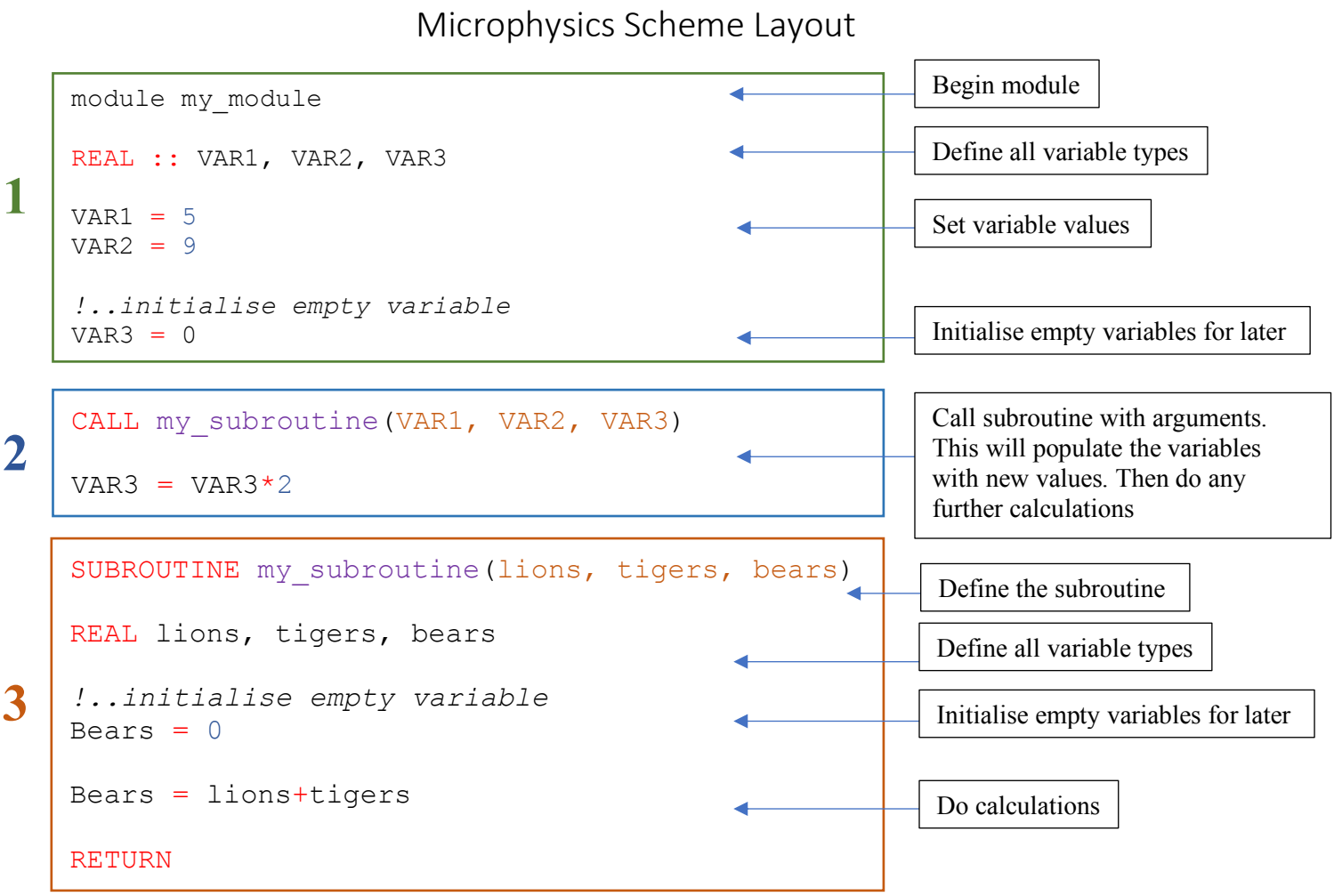
The SUBROUTINE in Fortran is slightly different. The arguments it returns are the arguments it takes, so the output must be defined at the input stage.

The variable `type` must be defined before any computation in Fortran. For example, `input_one`, `input_two` and `output` are defined as `REAL` (decimal, 32 bit precision). Whereas in python, the variable type is inferred by the interpreter so `output` will be a `REAL` because the inputs are also `REAL`.

Notice that a python function returns a specific value `output` but the Fortran subroutine returns the updated input variables to system memory.

Finally, when a Fortran subroutine is called is requires a `CALL` . In python the function must be defined before a function call, whereas in Fortran the subroutine is usually found below the call. This is because Fortran code is compiled before execution.

With this information we can see the layout of the subroutines:



In general, all microphysics schemes tend to follow this layout. Sometimes there are more subroutines that initialise the scheme and will run once in the first timestep. To extract a variable you must move it from the *lowest level* to the *top-level*.

For example, lets trace the path (or thread) of the variable `lions` in box 3. It is defined in `my_subroutine` and returned as an argument, it is fed into system memory when we issue a `call` and is named `VAR1`. So, at the top-level the variable is available to be extracted from `my_module`. You must follow this method when extracting a variable from within subroutines.

2. Modules

In Fortran, a module contains a collection of subroutines and functions. Modules are useful because they can be loaded by other modules and shared amongst the code hierarchy. For example, suppose that we want to use `my_subroutine` (above), then we can simply load `my_module` in a separate file and execute that file.

In WRF, each microphysics scheme occupies its own module. Hence, all of the microphysics schemes are saved in Fortran files (.F extension) with the name "`module_mp_XXXX`" where XXX is the microphysics scheme name or authors.

The microphysics scheme is called by `module_microphysics_driver.F`, which loads all microphysics modules, and applies the specific microphysics scheme from the namelist settings. You might write the path like:

`My_variable` → `module_mp_myscheme.F` → `module_microphysics_driver.F`

Or

```
module_microphysics_driver.F
      → module_mp_myscheme.F
            →My_variable
```

The hierarchy of calls in the WRF model are:

```
solve_em.F → subroutine solve_em → CALL microphysics_driver
       →module_microphysics_driver.F→ subroutine microphysics_driver →CALL MP_MORR_TWO_MOMENT
              →module_mp_morr_two_moment.F → subroutine MP_MORR_TWO_MOMENT → CALL MORR_TWO_MOMENT_MICRO
```

Where we have used the Morrison scheme as an example. Note that on each line a subroutine is defined that calls a subroutine defined lower in the code hierarchy. These modules and subroutines are the 'eyes' of the boot that we must thread our variable through.

Once our variable has been threaded all the way to `solve_em.F` we can reference it in the registry. The WRF registry is a collection of variable names for output with some metadata that will be written to file. For example:

Table	Type	Sym	Dims	Use	NumTLev	Stagger	IO	DNAME	DESCRIP	UNITS
state	real	SNOWNC	ij	misc	1	-	h	"SNOWNC"	"ACCUMULATED TOTAL GRID SCALE SNOW AND ICE"	"mm"

Where we will only edit the underlined columns.

3. Step by Step

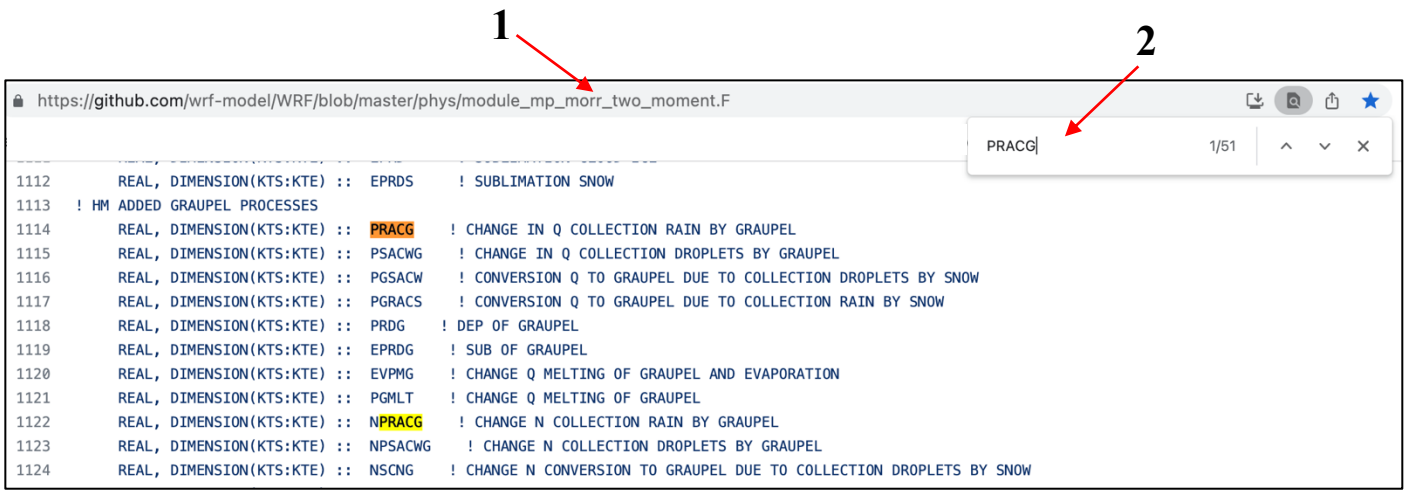
I recommend that you search for the thread of your variables using a browser such as chrome and the WRF GitHub pages (https://github.com/wrf-model/WRF/). This will help you to quickly navigate through files.

In this example we will extract a variable that is already defined in the code (i.e. we will not be making a new calculation). In this case I will use the Morrison scheme. Suppose that I want to output the variable **PRACG**, which is the microphysical **P**rocess of **R**ain **Ac**cretion to **G**raupel.

1. Navigate to the Fortran file of your microphysics scheme.

As I am using Morrison I will navigate to *WRF/blob/master/phys/module_mp_morr_two_moment.F*

2. Find your variable within the microphysics scheme subroutine definition by searching for it.




```
1112       REAL, DIMENSION(KTS:KTE) ::  EPRDS     ! SUBLIMATION SNOW
1113   ! HM ADDED GRAUPEL PROCESSES
1114       REAL, DIMENSION(KTS:KTE) ::  PRACG    ! CHANGE IN Q COLLECTION RAIN BY GRAUPEL
1115       REAL, DIMENSION(KTS:KTE) ::  PSACWG    ! CHANGE IN Q COLLECTION DROPLETS BY GRAUPEL
1116       REAL, DIMENSION(KTS:KTE) ::  PGSACW    ! CONVERSION Q TO GRAUPEL DUE TO COLLECTION DROPLETS BY SNOW
1117       REAL, DIMENSION(KTS:KTE) ::  PGRACS    ! CONVERSION Q TO GRAUPEL DUE TO COLLECTION RAIN BY SNOW
1118       REAL, DIMENSION(KTS:KTE) ::  PRDG    ! DEP OF GRAUPEL
1119       REAL, DIMENSION(KTS:KTE) ::  EPRDG    ! SUB OF GRAUPEL
1120       REAL, DIMENSION(KTS:KTE) ::  EVPMG    ! CHANGE Q MELTING OF GRAUPEL AND EVAPORATION
1121       REAL, DIMENSION(KTS:KTE) ::  PGMLT    ! CHANGE Q MELTING OF GRAUPEL
1122       REAL, DIMENSION(KTS:KTE) ::  NPRACG    ! CHANGE N COLLECTION RAIN BY GRAUPEL
1123       REAL, DIMENSION(KTS:KTE) ::  NPSACWG    ! CHANGE N COLLECTION DROPLETS BY GRAUPEL
1124       REAL, DIMENSION(KTS:KTE) ::  NSCNG    ! CHANGE N CONVERSION TO GRAUPEL DUE TO COLLECTION DROPLETS BY SNOW
```

3. Find the Containing Subroutine.

A good starting point is to find where the variable is calculated, in this case **PRACG(k)** is calculated on line 1765, where **k** means that this is a column array.

```
1763
1764   ! PRACG IS MIXING RATIO OF RAIN PER SEC COLLECTED BY GRAUPEL/HAIL
1765               PRACG(K) = CONS41*(((1.2*UMR-0.95*UMG)**2+                    &
1766                     0.08*UMG*UMR)**0.5*RHO(K)*                          &
1767                     N0RR(K)*N0G(K)/LAMR(K)**3*                           &
1768                     (5./(LAMR(K)**3*LAMG(K))+                          &
1769                     2./(LAMR(K)**2*LAMG(K)**2)+                          &
1770                                   0.5/(LAMR(k)*LAMG(k)**3)))
1771
```

So we know that the subroutine definition must be above line 1765. If I search SUBROUTINE then I can find the nearest subroutines and check if PRACG is defined within. In my case, the subroutine MORR_TWO_MOMENT_MICRO begins on line 929 and ends on line 4062 with no subroutines in between these lines. Therefore, we know that PRACG is defined within this subroutine.

4. Check that our variable is in the subroutine arguments.

Below is an image of the CALL to the subroutine MORR_TWO_MOMENT_MICRO that contains our variable. Recall that a subroutine updates its arguments when called. Therefore, if we want to extract

```
819
820         call MORR_TWO_MOMENT_MICRO(QC_TEND1D, QI_TEND1D, QNI_TEND1D, QR_TEND1D,           &
821          NI_TEND1D, NS_TEND1D, NR_TEND1D,                                                 &
822          QC1D, QI1D, QS1D, QR1D,NI1D, NS1D, NR1D,                                         &
823          T_TEND1D,QV_TEND1D, T1D, QV1D, P1D, DZ1D, W1D, WVAR1D,                  &
824          PRECPRT1D,SNOWRT1D,                                                              &
825          SNOWPRT1D,GRPLPRT1D,                 & ! hm added 7/13/13
826          EFFC1D,EFFI1D,EFFS1D,EFFR1D,DT,                                                  &
827                                               IMS,IME, JMS,JME, KMS,KME,                  &
828                                               ITS,ITE, JTS,JTE, KTS,KTE,                  & ! HM ADD GRAUPEL
829                                     QG_TEND1D,NG_TEND1D,QG1D,NG1D,EFFG1D, &
830                                     qrcu1d, qscu1d, qicu1d, &
831    ! ADD SEDIMENTATION TENDENCIES
832                                    QGSTEN,QRSTEN,QISTEN,QNISTEN,QCSTEN, &
833                                    nc1d, nc_tend1d, iinum, C2PREC,CSED,ISED,SSED,GSED,RSED & !wrf-chem
834    #if (WRF_CHEM == 1)
835                                    ,has_wetscav,rainprod1d, evapprod1d & !wrf-chem
836    #endif
837                             )
838
839        !
840        ! Transfer 1D arrays back into 3D arrays
841        !
842           do k=kts,kte
```

PRACG from the subroutine it must be in the argument list.

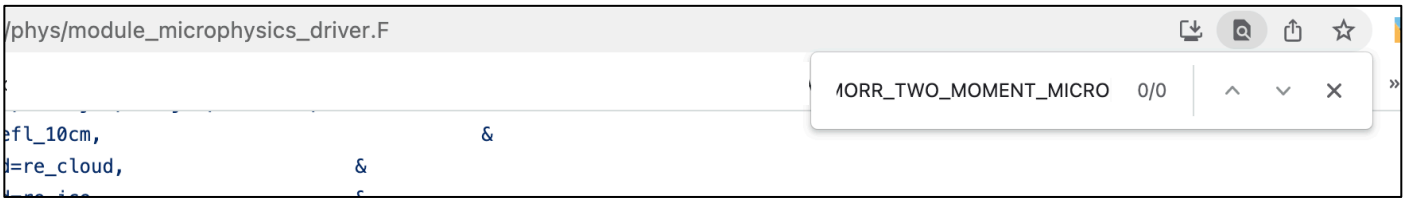
In the screenshot above I have highlighted the subroutine arguments (the layout isn't very clear). You will see that **PRACG** is not in the argument list. This means that the variable **PRACG** is local only, i.e. it is defined in the subroutine, calculated and lost with each call. The reason that this variable is not output is because it is used to form part of the mass tendency equation for graupel (**QGTEND1D**; line 829) and rain (**QR_TEND1D**; line 820).

As the variable is not a subroutine argument, we must modify the argument list to include **PRACG**. I strongly recommend inserting this in a consistent place. For example, at the start of the argument list. You could insert it at the end of the argument list but on line 834 there is an **IF** statement for WRF CHEM runs, so ensure that the variable goes before this **IF** statement.

Ensure that any change you make to a CALL argument list is mirrored in the SUBROUTINE argument list, and vice versa.

5. Check that it is called in the microphysics driver

It is likely that the microphysics-scheme file contains more than one subroutine and that the subroutine that calculates your variable is not the subroutine called by the microphysics driver. We can check this by examining the microphysics scheme file, or by examining the microphysics driver file.



See how I have searched the microphysics driver file for the subroutine but it does not exist. So the microphysics driver is actually calling something else. I can search for MORR or similar in the microphysics driver, or examine which subroutine contains the CALL to MORR_TWO_MOMENT_MICRO in the microphysics scheme file.

```
1500            CALL mp_morr_two_moment(                              &
1501                         ITIMESTEP=itimestep,                  &  !*
1502                         TH=th,                                &  !*
1503                         QV=qv_curr,                           &  !*
1504                         QC=qc_curr,                           &  !*
1505                         QR=qr_curr,                           &  !*
1506                         QI=qi_curr,                           &  !*
1507                         QS=qs_curr,                           &  !*
1508                         QG=qg_curr,                           &  !*
1509                         NI=qni_curr,                          &  !*
1510                         NS=qns_curr,                          &  !* ! VVT
1511                         NR=qnr_curr,                          &  !* ! VVT
1512                         NG=qng_curr,                          &  !* ! VVT
1513                         RHO=rho,                              &  !*
1514                         PII=pi_phy,                           &  !*
1515                         P=p,                                  &  !*
1516                         DT_IN=dt,                             &  !*
```

Either way, we find that the subroutine **mp_morr_two_moment** is called by microphysics driver. This might be called a *wrapper* or *interface* but its purpose is often the same amongst microphysics schemes: it iterates over the interior microphysics scheme for all grids in the domain, and constructs the 3D arrays for input/output.

In the Morrison case, all of the physical calculations are within the **MORR_TWO_MOMENT_MICRO** subroutine, which calculates column (k) arrays and returns key variables. The wrapper subroutine **MP_MORR_TWO_MOMENT** iterates over each column and each timestep in the grid, calls the MICRO subroutine, and assigns the output to a 4D array. The wrapper doesn't do a lot of the grubby complex work, but it is an essential stage for communication of the variables between the scheme and the broader WRF model output.

As in step 4, insert the variable (PRACG) into the subroutine arguments and calls. Ensure that the variable is located identically in each (order of variables is important!). You may then edit the argument call in the microphysics driver file with the format MYVAR=new_var_name. You don't have to change the name here, for example you may leave **PRACG** as:

PRACG=PRACG

Or change it to something else:

PRACG=my_pracg

In the former case you are less likely to be confused and can refer to your variables from the registry within the code. In the latter case, you must remember to reference with the new variable name from here in the

next steps but it may be useful to insert a subscript that identifies the variable as an edit of the original file (such as "**my_**"). Each method has its advantages, it is up to your personal preference.

6. What if my Variable is not the Right Dimension?

If you are extracting a variable for the first time (i.e. it is not predefined in the subroutine arguments or registry) then you will likely have to build the multi-dimensional array from the lesser-dimensional microphysics scheme output. For example, I want **PRACG** to be a three-dimensional array, but the microphysics subroutine only outputs **PRACG(k)** a column array. I need to save the columns after each call to a larger array.

The easiest thing you can do here is to copy how this process is already completed for an existing variable. Lets check the graupel mass mixing ratio, in the CALL to **MORR_TWO_MOMENT_MICRO** it is referred to as **QG1D** on line 829.

After the call, **QG1D** is next referenced on line 855 where the 3D array QG(I,K,J) is set equal to the column array.

```
839      !
840      ! Transfer 1D arrays back into 3D arrays
841      !
842         do k=kts,kte
843
844   ! hm, add tendencies to update global variables
845   ! HM, TENDENCIES FOR Q AND N NOW ADDED IN M2005MICRO, SO WE
846   ! ONLY NEED TO TRANSFER 1D VARIABLES BACK TO 3D
847
848              QC(i,k,j)        = QC1D(k)
849              QI(i,k,j)        = QI1D(k)
850              QS(i,k,j)        = QS1D(k)
851              QR(i,k,j)        = QR1D(k)
852              NI(i,k,j)        = NI1D(k)
853              NS(i,k,j)        = NS1D(k)
854              NR(i,k,j)        = NR1D(k)
855              QG(I,K,j)        = QG1D(K)
```

Note that this occurs for each cell in the column (line 842). Indeed, the whole call and column arrangement loop is nested in an iteration over lat/lon grid cells on lines 757-758 whereas timesteps are iterated higher in the code hierarchy.

The final step is to create a 3D array and nest it appropriately. Examine how this is done with existing variables and recreate with your own 3d variable (this will require a REAL definition and initialisation to 0). Ensure that the 3D variable is within the governing subroutine arguments.

7. Check that your variable is called in the **solve_em** file

Follow the steps above to trace the microphysics driver subroutine call in **solve_em.F**. At this stage, the variable can be fed to the grid:
PRACG =grid%**PRACG**

Once the variable is grid-referenced it is visible to the WRF registry.

8. Add it to the Registry

The WRF registry contains all of the variables in the grid and tells WRF which ones to write to the history file (output). First check that the variable you are extracting isn't already defined in the registry:

Next, find a microphysics variable that is defined in the registry. For example, cloud water radius:

```
496   state   real   re_cloud        ikj   misc   1   -   r   "RE_CLOUD"       "Effective radius cloud water"   "m"
497   state   real   re_ice          ikj   misc   1   -   r   "RE_ICE"         "Effective radius cloud ice"     "m"
498   state   real   re_snow         ikj   misc   1   -   r   "RE_SNOW"        "Effective radius snow"          "m"
499   state   real   re_cloud_gsfc   ikj   misc   1   -   rh  "RE_CLOUD_GSFC"  "Cloud Water effective radius"       "micron"
500   state   real   re_rain_gsfc   ikj   misc   1   -   rh  "RE_RAIN_GSFC"  "Rain Water effective radius"       "micron"
501   state   real   re_ice_gsfc    ikj   misc   1   -   rh  "RE_ICE_GSFC"   "Cloud Ice effective radius"        "micron"
```

We will copy this line and insert it again. The location isn't very important but it may be helpful to keep your variables together.

The columns are given below. I have edited some column entries to suit my new variable:

Table	Type	Sym	Dims	Use	NumTLev	Stagger	IO	DNAME	DESCRIP	UNITS
state	real	PRACG	ikj	misc	1	-	h	"PRACG"	"Rain mass to graupel mass"	"kg kg-1 s-1"

You must ensure that the IO column contains a "h" for **h**istory (output). It may contain other letters such as **r**estart, **i**nitial or **b**oundary if required.

Ensure that the dimensions match your output variable. For example, PRACG occurs in every grid cell so is a three-dimensional variable and would require "ikj".

Sym must match the variable name in the grid, you are free to name is however you please in the DNAME column. The DNAME is a string (enclose in "") that will be used when loading with NETCDF.

It may not seem important now but please add a clear description and accurate units. You will thank yourself in the future!

Finally, add the variable to the microphysics scheme declarations. This can be found at the bottom of the registry file. For Morrison the declaration is on line 2991:

```
2989   package   thompson         mp_physics==8    -   moist:qv,qc,qr,qi,qs,qg;scalar:qni,qnr;state:re_cloud,re_ice,re_snow
2990   package   milbrandt2mom    mp_physics==9    -   moist:qv,qc,qr,qi,qs,qg,qh;scalar:qnc,qnr,qni,qns,qng,qnh
2991   package   morr_two_moment  mp_physics==10   -   moist:qv,qc,qr,qi,qs,qg;scalar:qni,qns,qnr,qng;state:rqrcuten,rqscuten,rqicuten
2992   package   cammgmpscheme    mp_physics==11   -   moist:qv,qc,qi,qr,qs;scalar:qnc,qni,qnr,qns;state:rh_old_mp,lcd_old_mp,cldfra_old_mp,cldfra
```

There are three categories: moist, scalar, and state. As we are using state (column 1) you must add the variable name (NOT the DNAME) to the end of the state variable list.

9. Recompile

That is most of the coding done! However, none of these changes will take effect until we recompile WRF. Go into the WRF directory in your build and issue a **clean -a** then recompile. You do not need to recompile WPS.

If you have made an error it is likely that the compile will break and end abruptly. Check the compile output files for Error (with a capital E). Attempt to correct the error and recompile. This might take several attempts; to date I have never extracted a variable without causing at least one error so try not to be disheartened if this is a lengthy process.

If you are unlucky, then the compile will complete successfully only for the WRF code to break during simulation. Rinse and repeat the error finding process, remember to recompile each time you make changes.

Once you have completed a successful simulation, write your variable headers to an output file:

```
ncdump -h wrfout_d01_2013-02-08_12\:00\:00 > vars.txt
```

Search through the variables to ensure your variable name is present. If so, load the variable in python and check that it is not zero sized and appears consistent with the rest of the model results.

If your variable is not present you have made a mistake in steps 1-7, but most probably with the registry step. Check the *thread* of your variable carefully. A smart idea might be to perform your simulation for a short run (10 minutes) to ensure that your variable is properly output before launching into a large and lengthy simulation only to find it was a complete waste of time.

Hopefully now the metaphor becomes more clear. At each stage we thread the variable through subroutine definitions and calls, and through overarching files until we extract it!